Cryptography ECRYP
ElGamal Software implementation with ECB ciphering mode

Topor Dániel Bálint, Tulbure Angelo

January 18, 2022

Contents

1 Introduction **2**

2 Algorithm **3**
2.1 Key generation 3
2.2 Encryption 3
2.3 Decryption 3
2.4 Example 4
2.5 ECB 4

3 Code **5**
3.1 The code: 5
3.2 Testing 8
3.3 Behind the scenes 8

1 Introduction

Ciphers, also called encryption algorithms, are systems for encrypting and decrypting data. A cipher converts the original message, called plaintext, into ciphertext using a key to determine how it is done.

Ciphers are generally categorized according to how they work and by how their key is used for encryption and decryption. Block ciphers accumulate symbols in a message of a fixed size (the block), and stream ciphers work on a continuous stream of symbols. When a cipher uses the same key for encryption and decryption, they are known as symmetric key algorithms or ciphers. Asymmetric key algorithms or ciphers use a different key for encryption/decryption.

This project concerns the ElGamal cipher. ElGamal cryptosystem can be defined as the cryptography algorithm that uses the public and private key concepts to secure communication between two systems. It can be considered the asymmetric algorithm where the encryption and decryption happen by using public and private keys. In order to encrypt the message, the public key is used by the client, while the message could be decrypted using the private key on the server end. This is considered an efficient algorithm to perform encryption and decryption as the keys are extremely tough to predict.

In addition, we implement 2 ciphering modes: ECB (Electronic Code Book) and CBC(Cipher Block Chaining).

Block ciphers work on a fixed-length segment of plaintext data, typically a 64- or 128-bit block as input, and outputs a fixed length ciphertext. The message is broken into blocks, and each block is encrypted through a substitution process. Where there is insufficient data to fill a block, the blank space will be padded prior to encryption. The resulting ciphertext block is usually the same size as the input plaintext block.

The Electronic Code Book (ECB) mode uses simple substitution, making it one of the easiest and fastest algorithms to implement. The input plaintext is broken into several blocks and encrypted individually using the key. This allows each encrypted block to be decrypted individually. Encrypting the same block twice will result in the same ciphertext being returned twice.

In Cipher Block Chaining (CBC) mode, the first block of the plaintext is exclusive-OR'd (XOR'd), which is a binary function or operation that compares two bits and alters the output with a third bit, with an initialization vector (IV) prior to the application of the encryption key. The IV is a block of random bits of plaintext. The resultant block is the first block of the ciphertext. Each subsequent block of plaintext is then XOR'd with the previous block of ciphertext prior to encryption, hence the term "chaining." Due to this XOR process, the same block of plaintext will no longer result in identical ciphertext being produced.

Decryption in the CBC mode works in the reverse order. After decrypting the last block of ciphertext, the resultant data is XOR'd with the previous block of ciphertext to recover the original plaintext.

The CBC mode is used in hash algorithms. Discarding all previous blocks, the last resulting block is retained as the output hash when used for this purpose.

2 Algorithm

In order to make it simpler and clearer we will use an example of Bob and Alice. We suppose that Alice wants to communicate with Bob. In ElGamal, only the receiver needs to create a key in advance and publish it. Let's look at Bob's procedure of key generation.

2.1 Key generation

To generate his private key and his public key Bob does the following:

- **Prime and group generation:** Bob needs to select a large prime p and the generator g of a multiplicative group Z_p^* of the integers modulo p.
- **Private key selection:** Bob selects an integer x from the group Z at random and with the constraint $1 \leq x \leq p - 1$.
- **Public key assembly:** We calculate the public key part $y = g^x (modp)$. In ElGamal, the public key of Bob is the triplet (p, g, y) and his private key is x.
- **Public key publishing:** Bob must give this public key to Alice using a dedicated key server or other means.

To encrypt a plaintext message to Bob, Alice needs to get the public key. Our private key x is sent in y. The assumption that it is infeasible to compute using discrete logarithm means that this is safe. Let's look at Alice's plaintext message encryption.

2.2 Encryption

To encrypt a message Alice uses Bob's public key :

- **Obtain public key:** Alice acquires public key (p, g, y) from Bob.
- **Prepare M for encoding:** Prepare message M (to send) as set of integers $m_1, m_2, \ldots$ in the range of $\{1, \ldots, p - 1\}$ These integers will be encoded one by one.
- **Select random exponent:** Alice selects a random exponent k that's takes place of second party's private exponent.
- **Compute public key:** To transmit k to Bob, Alice computes $a = g^k (modp)$ and combines it with the ciphertext to be sent to Bob.
- **Encrypt the plaintext:** Alice encrypts message M to ciphertext C. To do this, she iterates over $m_1, m_2, \ldots$ and for each m_i:
  $c_i = m_i * (g^x)^k \rightarrow c_i = m_i * y^k$

The ciphertext C is the set of all c_i with $0 \leq i \leq |M|$

The resulted message C is sent to Bob along with public key $a = g^k (modp)$.

If an attacker listens to this transmission and acquires the public key part g^x of Bob, he would still not be able to derive g^{x*k} due to the discrete logarithm problem. ElGamal advises to use a new random k for each of the single message blocks m_i, which would lead to much higher security.

2.3 Decryption

After receiving the encrypted message C and the randomized public key g^k, Bob must use the encryption algorithm to read plaintext message M. Let's look at Bob's ciphertext decryption algorithm:

- **Compute shared key:** ElGamal cryptosystem allows Alice to define a shared secret key without Bob's interaction. This is from Bob's private exponent x and Alice's random exponent k. The shared key is defined as follows: $(g^k)^{-x} =$ an inverse in G of g^{kx}.

- **Decryption:** For each ciphertext parts c_i in C, Bob computes the plaintext using $m_i = (g^k)^{-x} * c_i(modp)$

He can read the message M sent by Alice by combining m_i .

2.4 Example

This example is made in order to better understand how ElGamal cryptosystem works.

Key generation:

1. A selects prime $p = 7187$ and generator $g = 754$ of Z^*_{7187}.
2. A chooses private key $x = 147$ and computes g^x (mod p) $= 754147$ (mod 7187) $= 6966$
3. A sends public key ($p = 7187$, $g = 754$, $g^x = 6966$) to B.

Encryption:

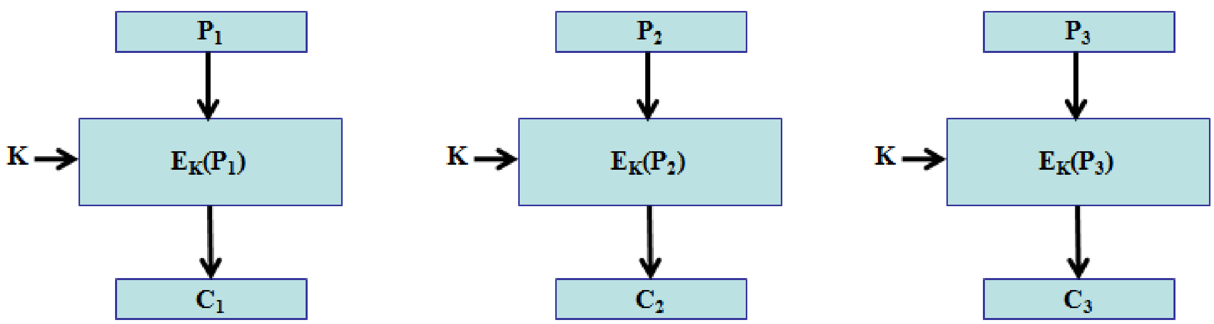
1. To encrypt message $m = 36$, B selects random integer $k = 55$.
2. B computes $a = g^k(modp) \rightarrow a = 75455$ (mod 7187) $\equiv 1571$ and $c = m * (g^x)^k \rightarrow 36 * 696655$ (mod 7187) $\equiv 6501$
3. B sends $a = 1571$ and $c = 6501$ to A.

Decryption:

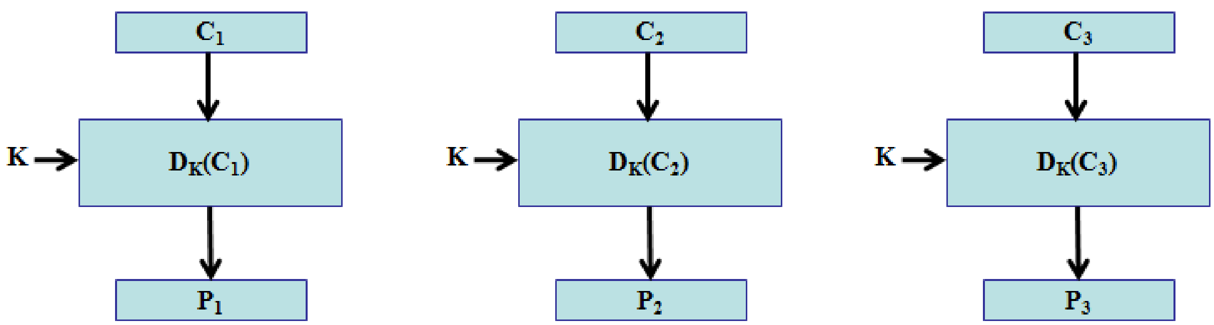
1. A computes $m_{plain} = c \;/\; (a^x)$ (mod p) $\rightarrow m = c \;/\; (g^k)^x$ (mod p) $\rightarrow m = (g^k)^{-x} * c$ (mod p) $\rightarrow 36$.

It satisfies the formula: $(g^k)^{-x} * c_i$ (mod p) $\equiv g^{-kx} * m * g^{xk} \equiv m$ (mod p), since $g^{-kx}\ g^{xk} = 1$.

2.5 ECB



a) Electronic codebook (ECB) encryption mode



b) Electronic codebook (ECB) decryption mode

We used the ECB mode in order to cipher our message. In the encryption part (ex: $E_k(P_2)$) we used ElGamal encryption and in the decryption part (ex: $D_k(C_1)$) we used ElGamal decryption.

3 Code

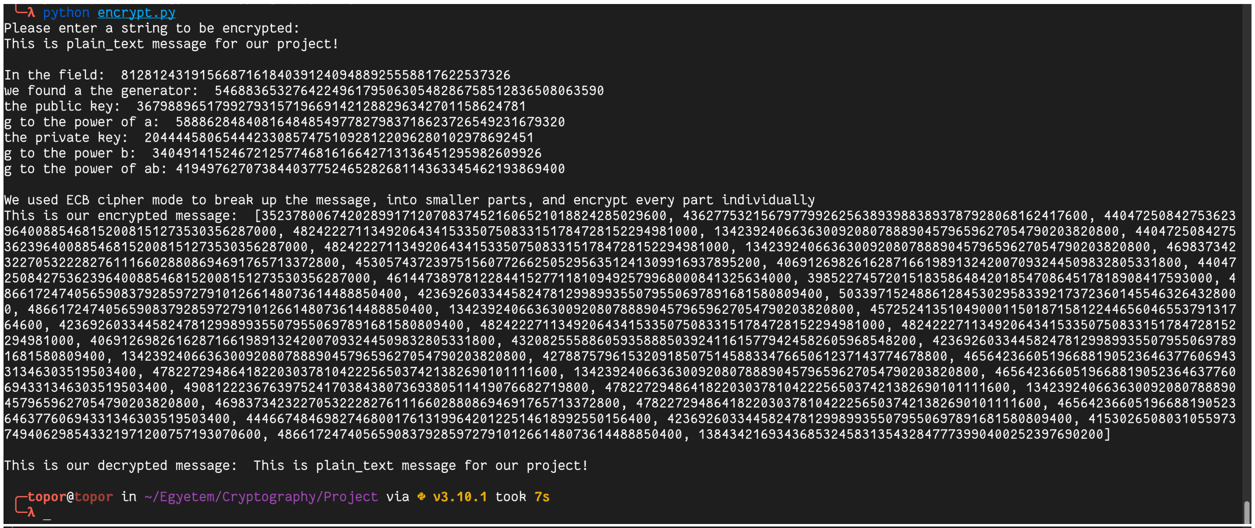
3.1 The code:

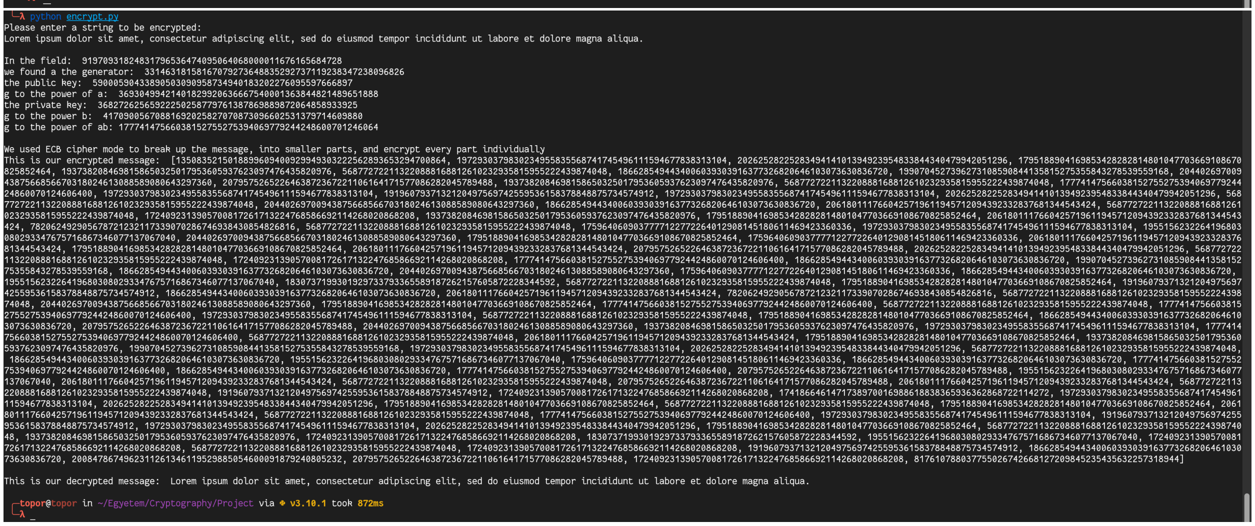
```
#!/usr/bin/env python3
import random
from math import pow

def power(a, b, c):
    x = 1
    y = a

    while b > 0:
        if b % 2 != 0:
            x = (x * y) % c;
        y = (y * y) % c
        b = int(b / 2)

    return x % c

def encrypted_elgamal(message, g_power_ab):
    return (g_power_ab * message)

def decrypted_elgamal(en_msg, g_power_ab):
    return int(en_msg / g_power_ab)

def encrypt_ECB(message, g_power_ab):
    c = []

    for i in range(0, len(message)):
        c.append(message[i])

    for i in range(0, len(c)):
        c[i] = encrypted_elgamal(ord(c[i]), g_power_ab)

    return c

def decrypt_ECB(en_msg, h):
    dr_msg = []

    for i in range(0, len(en_msg)):
        dr_msg.append(0)

    for i in range(0, len(en_msg)):
        dr_msg[i] = chr(decrypted_elgamal(en_msg[i], h))

    return dr_msg


def encrypt_CBC(plain_text, key, iv):
    c = []

    for i in range(0, len(plain_text)):
        c.append(1)
        c[i] = ord(plain_text[i]) #from char to int

    c[0] = c[0] ^ iv #scrable it
```

```
    for i in range(0, (len(c) - 1)):
        c[i] = encrypted_elgamal(c[i], key)
        c[i+1] = c[i+1] ^ c[i]

    c[len(c) - 1] = encrypted_elgamal(c[len(c) - 1], key)

    return c

def decrypt_CBC(cipher_text, key, iv, p):
    de_msg = []
    plain_text = []

    for i in range(0, len(cipher_text)):
        de_msg.append(0)
        plain_text.append('a')
        de_msg[i] = decrypted_elgamal(cipher_text[i], key)

    print("key = ", key, "ciphertext[2] = ", cipher_text[2])

    plain_text[0] = chr(iv ^ de_msg[0])
    print(0, plain_text[0])

    for i in range(1, len(cipher_text)):
        print(i, cipher_text[i-1], de_msg[i], cipher_text[i-1] ^ de_msg[i])
        plain_text[i] = chr(cipher_text[i-1] ^ de_msg[i])

    return plain_text

def gcd(a, b):
    if a < b:
        return gcd(b, a)
    elif a % b == 0:
        return b;
    else:
        return gcd(b, a % b)

# Generating large random numbers
def gen_key(p):

    key = random.randint(pow(10, 20), p)
    while gcd(p, key) != 1:
        key = random.randint(pow(10, 20), p)

    return key

def main():
    print("Please enter a string to be encrypted: ")
    message = input()

    p = random.randint(pow(10, 20), pow(10, 50))
    g = random.randint(2, p - 1)

    print("\nIn the field: ", p, "\nwe found a the generator: ", g)
    rec_key = gen_key(p) # Private key for receiver # a
    g_power_a = power(g, rec_key, p)
    print("the public key: ", rec_key)
    print("g to the power of a: ", g_power_a)

    send_key = gen_key(p)# Private key for sender # b
```

```
    g_power_b = power(g, send_key, p)
    print("the private key: ", send_key)
    print("g to the power b: ", g_power_b)

    g_power_ab = power(g_power_a, send_key, p)
    print("g to the power of ab:", g_power_ab, "\n")

    #print("Please choose a ciphering mode:\n1 ECB\n2 CBC")
    #cipher_mode = input()

    #if cipher_mode == "1":

    # ECB
    cipher_text = encrypt_ECB(message, g_power_ab)
    print("We used ECB cipher mode to break up the message, into smaller parts, and encrypt
        every part individually")
    print("This is our encrypted message: ", cipher_text, "\n")
    plain_text = decrypt_ECB(cipher_text, g_power_ab)
    dmsg = ''.join(plain_text)
    print("This is our decrypted message: ", dmsg)

    # else:
    #     # CBC
    #     iv = random.randint(1, 2000000)
    #     cipher_text = encrypt_CBC(message, g_power_ab, iv)

    #     plain_text = decrypt_CBC(cipher_text, g_power_ab, iv, p)
    #     dmsg = ''.join(plain_text)
    #     print(dmsg)

if __name__ == "__main__":
    main()
```

3.2 Testing





3.3 Behind the scenes

We commented out the CBC part, we tried to implement it, but we failed to decrypt it.